New Scheduling Strategies and Hybrid Programming for a Parallel Right-looking Sparse LU Factorization Algorithm on Multicore Cluster Systems

Ichitaro Yamazaki
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA 94720, U.S.A.
Email: ic.yamazaki@gmail.com

Xiaoye S. Li
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA 94720, U.S.A.
Email: xsli@lbl.gov

***Abstract*—Parallel sparse LU factorization is a key computational kernel in the solution of a large-scale linear system of equations. In this paper, we propose two strategies to address some scalability issues of a factorization algorithm on modern HPC systems. The first strategy is at the algorithmic-level; we schedule independent tasks as soon as possible to reduce the idle time and the critical path of the algorithm. We demonstrate using thousands of cores that our new scheduling strategy reduces the runtime by nearly three-fold from that of a state-of-the-art pipelined factorization algorithm. The second strategy is at both programming- and architecture-levels; we incorporate light-weight OpenMP threads in each MPI process to reduce both memory and time overheads of a pure MPI implementation on manycore NUMA architectures. Using this hybrid programming paradigm, we obtain a significant reduction in memory usage while achieving a parallel efficiency competitive with that of a pure MPI paradigm. As a result, in comparison to a pure MPI paradigm which failed due to the per-core memory constraint, the hybrid paradigm could utilize more cores on each node and reduce the factorization time on the same number of nodes. We show extensive performance analysis of the new strategies using thousands of cores of the two leading HPC systems, a Cray-XE6 and an IBM iDataPlex.**

I. Introduction

Parallel sparse LU factorization is widely used for solving a large-scale linear system of equations in scientific and engineering simulations. It can be used alone as a direct solver, or it can be used as a preconditioner for an iterative solver. However, implementing a parallel factorization algorithm that is scalable in both time and memory is a formidable task even for an expert in parallel computing. This is because such an algorithm possesses many of the fundamental challenges of parallel programming such as highly irregular memory access patterns, large degree of task and data dependencies, and imbalances in data distribution and workload. This situation is exacerbated by the modern HPC computers with heterogeneous manycore NUMA node architectures. In this paper, we propose two strategies to address some of the scalability issues of a parallel factorization algorithm: the first is at the algorithmic-level; we schedule independent tasks as soon as possible to reduce the idle time and the critical path of the algorithm. The second is at both programming- and architecture-levels; we employ a hybrid programming paradigm to fully utilize the node-level parallelism and memory of multicore NUMA architectures.

Our investigation focuses on a widely used open source library `SuperLU_DIST` [22], which is a package for the direct solution of a large-scale sparse general linear system of equations on a distributed-memory cluster. It is a state-of-the-art parallel direct solver capable of solving linear systems with millions of unknowns from real-world applications [5]. The original target of `SuperLU_DIST` was the earlier generations of distributed-memory systems, where each compute node had one or a small number of processor cores with the uniform access to the physical memory. On a modern HPC computer with the multicore NUMA node architecture, the parallel scaling of `SuperLU_DIST` often stagnates on a few hundred of cores. Performance profiling on 256 processor cores of the Cray-XE6 system at NERSC revealed that about 81% of the factorization time was spent in MPI_Wait() and MPI_Recv(). In other words, for the 81% of the time, the processor cores were performing neither computation nor communication. To reduce this idle time, in this paper, we propose an algorithmic-level strategy to statically schedule independent tasks as soon as possible. Our experimental results will demonstrate that the parallel factorization with this new scheduling strategy can obtain speedups of up to three over the current version of `SuperLU_DIST` which is based on a pipelined factorization [22].

The Cray-XE6 system at NERSC is a representative of the new generation of a multicore NUMA architecture. Each node of this system has two tweleve-core MagnyCours processors and 32GB of memory, averaging about 1.3GB of memory per core. Even though our aforementioned scheduling strategy shortens the critical path of the algorithm, there are two hindering factors for `SuperLU_DIST` to fully utilize all the cores on each node. The first factor is the per-core memory constraint. The current `SuperLU_DIST` is based on a pure MPI programming paradigm, where the increase in the number of MPI processes often increases the total memory requirement. This is because the total communication volume of the algorithm often increases with an increase in the number of MPI processes, and moreover, each message may be internally duplicated in multiple communication buffers by an MPI implementation. Finally, SuperLU_DIST has certain amount of serial memory overhead associated with an MPI process (see Section III). As a result, to solve a large-scale linear system under the per-core memory constraint, SuperLU_DIST can often use only a limited number of cores on each node. On the future computer with hundreds or thousands of cores per node, the per-core memory is expected to be even smaller [6]. The second factor is that on a multicore architecture, a message-passing paradigm often has a greater time overhead than a shared-memory paradigm. Therefore, even when the per-core memory constraint did not hinder the usage of all the cores on each node, the factorization time may not scale. Hence, it is imperative to abandon the pure MPI paradigm and resort to a certain type of a hybrid programming paradigm that can exploit the NUMA architecture. To demonstrate this, in the second part of this paper, we incorporate a hybrid message-passing (MPI) and shared-memory (OpenMP) programming paradigm into SuperLU_DIST. Using this hybrid paradigm, we obtained a significant reduction in memory usage while achieving a parallel efficiency competitive to that of a pure MPI paradigm. As a result, in comparison to a pure MPI paradigm, the hybrid paradigm could utilize more cores on each node and reduce the factorization time on the same number of nodes.

The rest of the paper is organized as follows. In Sections II and III, we first discuss related works and give a brief overview of SuperLU_DIST, respectively. Then, in Section IV, we describe two techniques, look-ahead and static scheduling, which are designed to reduce the idle time during the parallel factorization. Next, in Section V, we discuss our attempt to incorporate a hybrid MPI+OpenMP programming paradigm into SuperLU_DIST; The performance results in Section VI will demonstrate that these techniques can significantly improve the performance of SuperLU_DIST on leading HPC computers based on multicore NUMA node architectures. We conclude with final remarks in Section VII.

II. Related work

There have been several scheduling strategies and hybrid programming paradigms proposed for parallel sparse direct solvers. In this section, we briefly describe those that are most relevant to the ones proposed in this paper.

PasTiX [15] implements a parallel left-looking supernodal factorization algorithm based on a hybrid MPI+pthread programming paradigm [16]. It is capable of solving both symmetric and unsymmetric systems, but it is most effective for solving a linear system with a symmetric positive define (SPD) coefficient matrix A. It uses a combination of a static and dynamic scheduling schemes based on both elimination tree of $|A|^T + |A|$ and performance models capturing both computation and communication [9]. Another relevant solver is WSMP [13], [14] which implements multifrontal factorization algorithm for solving SPD and unsymmetric systems. It uses a hybrid MPI+pthreads programming paradigm, and an assembly tree (elimination tree) for scheduling. More recently, Hogg et al. used a dynamic scheduler for a shared-memory supernodal algorithm to factorize an SPD matrix [17]. The dependencies among the tasks are represented by an implicit direct acyclic graph (DAG), and the dependencies are resolved by keeping track of the outstanding incoming edges at runtime.

In comparison to these previous works, we focus on SuperLU_DIST which implements a supernodal right-looking LU factorization algorithm for solving general sparse linear systems. We first propose a static scheduling strategy which uses one of the following two underlying graphs to represents the task dependencies: the symmetrically pruned DAG of the LU factors and the elimination tree of $|A|^T + |A|$. We show that our scheduling strategy has very little runtime overhead on a large-scale multicore clusters and can significantly reduce the factorization time. We then study MPI+OpenMP hybrid paradigm to further enhance the performance of SuperLU_DIST.

III. Overview of SuperLU_DIST

To compute the solution of a sparse linear system, SuperLU_DIST first computes an LU factorization of the coefficient matrix, and then applies the forward and backward substitutions. The LU factorization typically dominates the solution time and is carried out in the following three steps:

1) Matrix pre-processing: Before the numerical factorization, the coefficient matrix A is first pre-processed to achieve two goals. The first goal is to enhance the numerical stability through static pivoting and matrix equilibration; i.e, we compute a row permutation matrix P_r, and a row and column equilibration matrices D_r and D_c. The serial code MC64 developed by Duff and Koster [7], which implements a maximum weighted matching algorithm, is employed. The algorithm computes P_r to maximize the product of the diagonal entries, and it also computes D_r and D_c simultaneously so that the nonzero diagonal entries of $P_r D_r A D_c$ are one in their absolute values and all the off-diagonal entries are less than or equal to one in their absolute values. It has been shown that these pre-processing techniques make the LU factorization numerically as stable as that using partial pivoting for a wide range of problems [21]. Hence, SuperLU_DIST does not employ dynamic pivoting (e.g., partial pivoting) during the numerical factorization.

The second goal of the pre-processing is to symmetrically reorder the matrix $P_r D_r A D_c$ such that its LU factors remain sparse. This reduces the computational and storage costs of the LU factorization. The reordering also helps

```
for $k = 1, 2, \dots, n_s$ do
  1. Panel factorization
     1.1 Column computation of $L(:, k)$.
        a. if $p_{\text{id}} \in P_C(k)$ then
        b.   compute the block column $L(k : n_s, k)$
             (communicate $U(k, k)$ among $P_C(k)$)
        c.   send $L(k : n_s, k)$ to required processes in $P_R(:)$
        d. else
        e.   receive $L(k : n_s, k)$ if required
        f. end if
     1.2 Row computation of $U(k, :)$.
        a. if $p_{\text{id}} \in P_R(k)$ then
        b.   wait for $U(k, k)$
        c.   compute the block row $U(k, k+1 : n_s)$
        d.   send $U(k, k+1 : n_s)$ to required processes in $P_C(:)$
        e. else
        f.   receive $U(k, k+1 : n_s)$ if required
        g. end if
  2. Outer-product updates of trailing submatrix.
     a. for $j = k+1, k+2, \dots, n_s$ with $U(k, j) \neq \emptyset$ do
     b.   for $i = k+1, k+2, \dots, n_s$ with $L(i, k) \neq \emptyset$ do
     c.     if $p_{\text{id}} \in P_R(i) \cap P_C(j)$
     d.       $A(i, j) \leftarrow A(i, j) - L(i, k)U(k, j)$
     e.     end if
     f.   end for
     g. endfor
end for
```

Figure 1. Numerical factorization algorithm in SuperLU_DIST.

to reduce communication and improve the load balance of numerical factorization [4]. Such a matrix ordering can be computed, for example, using a nested dissection algorithm of METIS [18] on the sparsity structure of $|P_r A|^T + |P_r A|$. For the remaining of this paper, we use A to denote the matrix after the pre-processing is applied.

2) Symbolic factorization: The main benefit of static pivoting over dynamic pivoting is to permit a priori determination of the sparsity structures of the LU factors before the numerical factorization. An efficient symbolic factorization algorithm [11], [21], [23] has been developed to determine the sparsity structure, set up the required data structures, and schedule all the communication and computation for the numerical factorization. This often makes SuperLU_DIST more scalable than the other solvers based on dynamic pivoting [4].

3) Numerical factorization: The numerical factorization is based on a fan-out (right-looking, outer-product) supernodal LU factorization algorithm. A supernode is a set of consecutive columns of L with a dense triangular block just below the diagonal and with the same nonzero structure below the triangular block. To achieve good parallelism and load balance, the MPI processes are assigned to the supernodal blocks in a 2D cyclic layout. Figure 1 shows the pseudocode of the factorization algorithm, where n_s is the number of supernodes, p_{id} is the ID of this process, and $P_C(k)$ and $P_R(k)$ are the groups of processes assigned to the k-th supernodal column and the k-th supernodal row, respectively. Step 1 of the pseudocode corresponds to the k-th panel factorization, where the k-th supernodal column of L and the k-th supernodal row of U are computed. At Steps 1.1.c and 1.2.d, each process in $P_C(k)$ and $P_R(k)$ sends its local blocks of the factors to the processes assigned to the same row and column, respectively. Then, Step 2 updates the trailing submatrix using the k-th supernodal column and row of the LU factors. To take advantage of the sparsity of A, the block $A(i, j)$ is updated only if both blocks $L(i, k)$ and $U(k, j)$ are not empty. More detailed description of the algorithm can be found in [22]

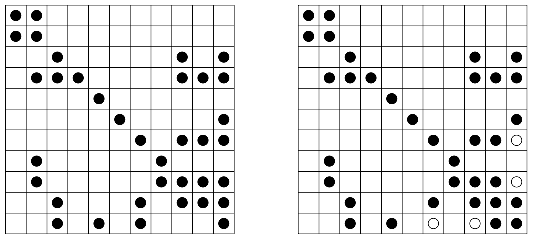

(a) Coefficient matrix A. (b) LU factors of A.

Figure 2. Nonzero patterns of a matrix A and its LU factors.

IV. New static task scheduling strategy

The factorization algorithm in Figure 1 follows a sequential flow, i.e., the panel factorizations and the trailing-submatrix updates are performed in sequence. For instance, the MPI processes in $P_C(k)$ and $P_R(k)$ must wait for the k-th diagonal block to be factorized before starting its panel factorization. Moreover, all the processes must wait for the panel factorization to complete before updating the trailing-submatrix. On the other hand, at each step, multiple panels may be ready to be factorized since they will not be updated by the remaining panels due to the sparsity of the matrix. Since several MPI processes may be idle waiting for the k-th panel factorization to complete, these MPI processes can be used to factorize the rest of the ready-to-be-factorized panels and reduce the idle time. Furthermore, by factorizing and sending these panels as soon as possible, their computation and communication can overlap with other computation and communication. In this section, we describe a new task scheduling strategy to exploit these parallelism that are not fully exploited in Figure 1.

A. Task dependency graph of sparse factorization

In this section, we introduce the task dependency graph of sparse LU factorization, which is an important tool for developing our scheduling algorithm. We will use the 11×11 supernodal matrix shown in Figure 2 for illustration, where each column and row of the matrix represent a supernodal column and row, respectively. Now, consider Step 2 of the factorization algorithm in Figure 1. We see that the j-th column is updated by the k-th column only if the block $U(k,j)$ is not empty. Similarly, the i-th row is updated by the k-th row only if the block $L(i,k)$ is not empty. These dependencies can be represented by a directed graph, where the k-th node represents the k-th panel factorization, and for each k-th node, there is a directed edge (k,j) or (k,i) for each non-empty block $U(k,j)$ or $L(i,k)$, respectively. The edge (k,j) (or (k,i)) represents the dependency that the k-th column (or row) updates the j-th column (or the i-th row). Figure 3 shows the dependency graph of the 11×11 matrix in Figure 2.

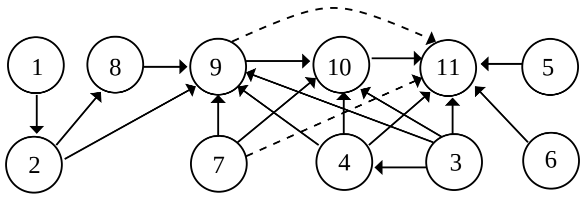
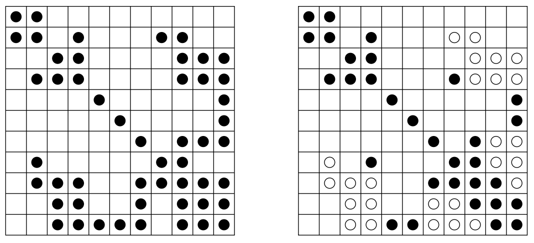


Figure 3. Dependency graph of LU factorization.

Unfortunately, it is usually not efficient to use the graph in Figure 3 as a scheduling tool. This is because it contains excessive amount of redundant information. For example, there are an edge $(7,10)$ and a path $7 \rightarrow 9 \rightarrow 10$. Hence, the edge $(7,10)$ is redundant.

A transitive reduction of a directed graph encompasses all the dependency information with the minimum number of edges [3]. However, its construction can be expensive. One alternative is to use a so-called symmetrically pruned graph [8]. To construct the pruned graph, we first identify the smallest index s_k such that $U(k,s_k)$ and $L(s_k,k)$ are the first symmetrically matched non-empty blocks for each k. Then, we prune all the edges (k,j) for $j > s_k$. The white circles in the matrix of Figure 2(b) and the dashed edges in the graph of Figure 3 represent the pruned edges. From now on, we refer to this symmetrically pruned graph as the reduced directed acyclic graph, or rDAG in short, of the LU factors. A node of the rDAG without any incoming edges is referred to as a source, while a node without any outgoing edges is called a sink. A similar task graph was used in [12] for a left-looking LU factorization algorithm, where only the column dependency needed to be enforced. On the other hand, our k-th panel factorization task factorizes both k-th row and column, and our task graph must keep track of both column and row dependencies.

For a symmetric matrix A, its rDAG is identical to its transitive reduction. Furthermore, in this case, rDAG is a tree, which is commonly referred to as an elimination tree, or etree in short, and is used extensively to study the behavior of sparse factorizations [24]. Just like rDAG, the k-th node of the etree represents our k-th panel factorization. There is an edge (k,j) from the k-th node (a child) to the j-th node (the parent) if $U(k,j)$ is the first non-empty block in the k-th row of the U-factor. In the etree, a node without



(a) Symmetrized matrix $|A|^T + |A|$. (b) LU factors of $|A|^T + |A|$.

Figure 4. Nonzero patterns of $|A|^T + |A|$ and its LU factors.

incoming edges is called a leaf, and the node without outgoing edges is referred to as a root. Even for the LU factorization of an unsymmetric matrix A, the etree of the symmetrized matrix $\widehat{A} = |A|^T + |A|$ can be used to capture both column and row dependencies of the panel factorizations. However, this etree of the symmetrized matrix can overestimate the true dependency of the panels in the actual unsymmetric factorization. On the other hand, rDAG contains some redundant edges, but it does not overestimate the dependency unless numerical cancelation occurs during the numerical factorization. Figures 4 and 5 respectively show the sparsity structure and etree of the symmetrized matrix $\widehat{A}$ of the 11×11 matrix A in Figure 2. In comparison to the rDAG in Figure 3, the etree in Figure 5 greatly overestimates the dependency of the panels, where the critical path of the etree is of length six while that of rDAG is of length three.

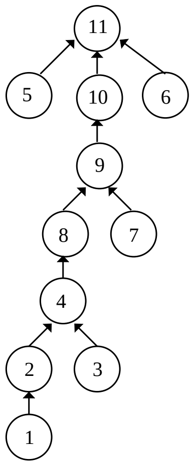


Figure 5. The etree of $|A|^T + |A|$.

For scheduling the panel tasks, the final LU factors will be correct as long as the following task-dependency invariant is preserved: before the j-th panel factorization, all the preceding updates to the j-th column and row must be completed. In other words, before scheduling the j-th task,

```
0. Initialize look-ahead window
   a. set $n_w$ (look-ahead window size)
   b. $n_0 = 1$ (index of the next column in window)
for $k = 1, 2, \ldots, n_s$ do
1. Look-ahead the new columns in the window.
   a. for $j = n_0, \ldots, k + n_w$ do
   b.     Panel factorize $A(j : n_s, j)$ if possible
          (communicate $U(j, j)$ among $P_C(j)$, and
           isend $L(j : n_s, j)$ to $P_R(:)$)
   c. end for
   d. $n_0 = k + n_w + 1$
2. Look-ahead the rows.
   a. for $i = k + 1, \ldots, k + n_w$ do
   b.     Factorize $A(i, i : n_s)$ if $U(i, i)$ has arrived
          (isend $U(i, i : n_s)$ to $P_C(:)$)
   c. end for
3. Wait for $U(k, k)$ and factorize $U(k, :)$ if needed.
4. Wait for $U(k, k : n_s)$ and $L(k : n_s, k)$.
5. Look-ahead factorization
   a. for $j = k + 1, \ldots, k + n_w$ with $U(k, j) \neq 0$ do
   b.     Update $A(j : n_s, j)$
   c.     Panel factorize $A(j : n_s, j)$ if possible
          (communicate $U(j, j)$ among $P_C(j)$, and
           isend $L(j : n_s, j)$ to $P_R(:)$)
   d. end for
6. Update the remaining trailing matrix.
end for
```

Figure 6. Pseudocode of look-ahead factorization.

all the tasks corresponding to the nodes in the dependency graph, which can reach the j-th node following the directed paths, must be completed.

Since our panel factorization factorizes both column and row, from now on, when we say that a column is factorized, the corresponding row is also factorized. Our discussion will focus on the etree, but will comment on how it can be extended to the rDAG.

B. Look-ahead

Using the aforementioned task dependency graph, we can schedule the panel factorization tasks in an order from the leaves to the root of the etree or from the sources to the sink of the rDAG. After the k-th panel updates the j-th column and row, the corresponding edge (k, j) is removed from the graph, potentially making the j-th node a leaf or source. These leaf-nodes represent the columns and rows that can be factorized. If we factorize all the leaf-nodes and asynchronously send the results to the trailing submatrix before updating the remaining submatrix, the idle time of the processes may be minimized. Unfortunately, factorizing and asynchronously sending all the leaf-nodes may require infeasibly large memory to store the pending messages.

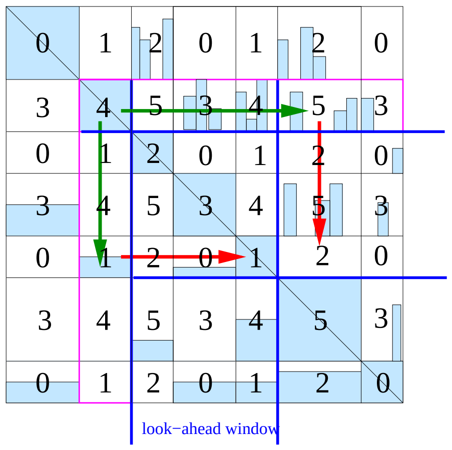


Figure 7. Illustration of look-ahead factorization.

To reduce the memory requirement, we look-ahead only a few next supernodal columns in a so-called look-ahead window of size n_w and see if they can be factorized. Specifically, if the j-th node in the look-ahead window becomes a leaf after the removal of the edge (k, j); i.e., this is the last update on the j-th column, then after the j-th column is updated, we immediately factorize the j-th column and send it to the trailing submatrix. This process is applied to all the columns with the non-empty block $U(k, j)$ in the look-ahead window (i.e., $j = k + 1, k + 2, \ldots, k + n_w$). Finally, the remaining columns outside the look-ahead window are updated as before. Figure 6 shows the pseudocode of the look-ahead algorithm, and Figure 7 illustrates the algorithm.

If $n_w > 1$, then the k-th column is factorized before the k-th step. This is because before the end of the $(k-1)$-th step, all the dependencies on the k-th column are removed, and since the k-th column is in the look-ahead window at the $(k-1)$-th step, it is factorized. Hence, at the beginning of the k-th step, if the block $U(k, j)$ is not empty, then the k-th column can be used right away to update the j-th column. Note that at the beginning of the k-th step, we first check if the $(k + n_w)$-th column, which was not in the look-ahead window during the $(k-1)$-th step, is already a leaf.

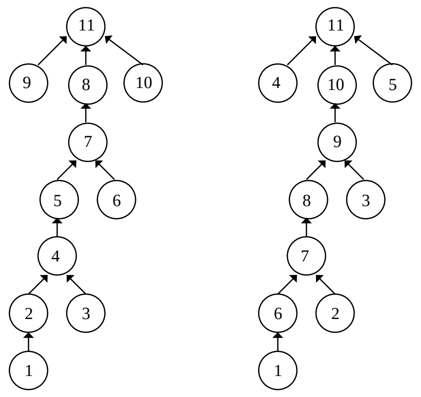
We now describe how we look-ahead supernodal rows. Let us assume that the j-th node in the look-ahead window becomes a leaf after the edge (k, j) is removed. Hence, the j-th column is factorized right after being updated with the k-th panel. On the other hand, the j-th row cannot be factorized, yet, if a block $U(j, \ell)$ for $\ell > j$ in the j-th row needs to be updated. This is why the rows in the look-ahead window are factorized separately from the columns. Specifically, when the j-th node becomes a leaf, the corresponding diagonal process (e.g., process 3 on the 4-th diagonal block in Figure 1) first factorizes its supernodal blocks in the j-column, and then sends the diagonal block to the processes in the same column (e.g., process 0). While the diagonal process performs its panel factorization, the rest of the processes in the j-th column are blocked. As soon as these processes in the column receive the diagonal block, they perform the panel factorization of their local blocks in the j-th column and send the results to the processes in the same row (e.g., process 3 sends to processes 5, and process 0 sends to process 2. Note that the results are sent only to the processes that require them; i.e., to the processes in the columns with the non-empty blocks in the j-th row). On the other hand, the panel factorization of the rows is implemented using non-blocking communication, and the processes perform the panel factorization of the row only after all the trailing updates with the k-th panel are completed and when the diagonal block is received (Step 2 of Figure 6). Hence, the process blocks only at the k-th step if the k-th diagonal block $U(k,k)$ has not been received, yet (Step 3).

This look-ahead technique has been used for dense matrix factorization [19], where the speedup of about 1.7 was reported on a shared-memory computer with two 1.8GHz dual-core AMD opteron 265 processors. SuperLU_DIST already implements a pipelining mechanism, where the next $(k+1)$-th column is factorized before the remaining columns are updated. This is equivalent to look-ahead with the window size of one. In [22], pipelining reduced the factorization time by 10% to 40% on 64 processors of a Cray-T3E system. Here, we generalize this idea to an arbitrary look-ahead window size, which allows higher degree of parallelism and overlapping of communication and computation.

C. Bottom-up topological ordering

The look-ahead mechanism in Section IV-B provides a great potential to reduce some serialization in SuperLU_DIST. However, even after the integration of look-ahead, we observed that on 256 cores of Cray-XE6, about 76% of the numerical factorization time was still spent at the synchronization points (e.g., Steps 3 and 4 of Figure 6). This is because even though many of the panel factorization tasks were leaves, they were outside the look-ahead window. Since these tasks enter the look-ahead window from the first to the n_s-th task in the sequence, the ordering of these tasks has a significant impact on the performance of look-ahead. We next propose an ordering of the supernodal columns to increase the potential of the tasks within the look-ahead window being leaves. Our main objective is to find an ordering of all these tasks as given in the outer loop over k in Figure 1 so that the critical path of the algorithm is shortened. Notice that this loop transformation is possible only for a sparse matrix A since its task dependency graph is not a complete graph; whereas the dependency graph of a dense matrix is complete.

Let us first discuss how SuperLU_DIST currently orders or schedules these tasks. The symbolic factorization algorithm permutes the columns of the coefficient matrix according to a postordering of the etree, in which the children are numbered before their parents and the nodes within any subtree are numbered consecutively (see Figure 8(a)).[1] This ordering is motivated to obtain larger supernodes without changing the sparsity structure of the LU factors. The reason this ordering may increase the sizes of the supernodes is the following. The sparsity structure of the fill generated in the j-th column is contained in the union of that of the j-th column of A and those of the columns of the L-factor, which correspond to the descendants of the j-th node in the etree [10]. Thus, after the matrix A is permuted in the postordering, the nodes corresponding to the adjacent columns are likely to have a large number of same descendants and are expected to have similar sparsity structures in the L-factor. As our symbolic factorization subroutine locates a supernodal column in the postorder, it sets up the data structure to store the column. Hence, these supernodal columns are stored at the contiguous memory locations in the postorder. Then, during the numerical factorization, the supernodal columns are factorized in the same postorder since this improves data locality of computing the LU factors. Unfortunately, this postordering limits the number of supernodal columns that can be factorized in the look-ahead window. This is because the look-ahead window contains only the nodes in a small subtree of the etree, while missing the other leaf-nodes in the other parts of the tree, which are ready to be factorized.



(a) postordering (b) bottom-up ordering

Figure 8. Static scheduling based on etree.

To mitigate this problem, we use a static scheduling scheme based on a bottom-up topological ordering of the etree, in the spirit of breadth-first search (see Figure 8(b)). This ordering can be computed using a FIFO queue. First, all the initial leaf-nodes in the etree are pushed into the queue (the nodes 1 through 5 in Figure 8(b)). Then, the first node in the queue is popped to be scheduled, and if the removal

[1]The nested dissection ordering is one example of postordering.

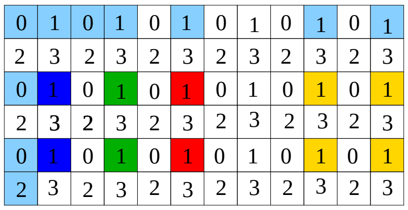
of this node generates a new leaf-node in the etree, then the new leaf-node is pushed into the end of the queue. If we use a priority-queue instead of a FIFO queue, then several options exist to schedule the leaf-nodes in the queue. In our implementation, to shorten the critical path of the algorithm, we try to schedule the leaf-node that is furthest away from the root first. This is done by ordering the initial leaf-nodes in the descending order of their distance from the root. Then, the new leaf-nodes are pushed into the FIFO queue as the nodes in the queue are processed (see Figure 8(b)).

For an unsymmetric matrix, we can either use the etree of the symmetrized matrix $|A^T| + |A|$ or use the rDAG by scheduling all the source-nodes of the rDAG first. With the combination of the static scheduling and look-ahead, only about 36% of the numerical factorization time is now spent at the synchronization points on the 256 cores of Cray-XE6. This scheduling strategy will be incorporated into the upcomming version 3.0 of `SuperLU_DIST`.

V. Hybrid programming

With the advent of multicore architecture, we are seeing an increasing number of cores per node and a simpler core design. In the near future, the number of cores per node is expected to be in the order of hundreds or thousands [6]. On the other hand, the size of the memory on each compute node is expected to be about the same or smaller due to power constraint. Using a pure MPI programming paradigm is not appropriate for such light-weight core designs, especially on NUMA architectures. For instance, the small amount of per-core memory can become a limiting factor for running one MPI process per core since each MPI process adds certain amount of communication buffer overhead. Even if we have sufficient memory to pack hundreds of MPI processes on each node, the network adapter on the node could become a serious bottleneck when many of these tasks communicate off-node. In order to effectively utilize the node-level core resources, the on-node parallel execution model must incorporate fine-grained data parallelism to reduce the message passing overhead. Hence, it becomes imperative to investigate new programming paradigms other than a pure MPI paradigm. In this section, we describe how we integrated a hybrid message-passing and shared-memory programming paradigm into `SuperLU_DIST` to adapt to the modern multicore cluster.

The computational cost of numerical factorization is typically dominated by the trailing submatrix update, where each process updates several independent blocks of the trailing submatrix at each step. We incorporated light-weight OpenMP threads in each MPI process to update disjoint sets of these independent blocks in parallel. We chose to use OpenMP over other threading or data parallel languages because it is production-ready, easily accessible, and widely supported.


(a) 1D block column layout

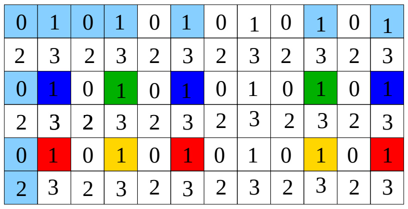

(b) 2D cyclic layout

Figure 9. Mapping of threads to supernodal blocks. The light-blue blocks represents the non-empty blocks in the current panel. Four MPI processes are assigned to blocks in a 2×2 grid, where the numbers inside the blocks indicate the process ID. Each MPI process generates four threads, where the blocks in blue, green, red and yellow are assigned to the first, second, third and fourth thread of the process 1, respectively. Only the active blocks are assigned to threads.

There are several options as to how to assign the independent blocks to the threads. For instance, a process can assign its local supernodal columns of the trailing submatrix to the threads in a 1D block fashion; i.e., the t-th thread updates $(t-1) \cdot h$-th to $(t \cdot h - 1)$-th columns, where $h = \frac{n_c}{n_t}$, n_t is the number of threads, and n_c is the number supernodal columns assigned to this process (see Figure 9(a)). Since these columns are contiguous in memory, each thread can access the columns without large stride. However, with this layout, the number of threads is limited by the number of columns. Another approach is to assign the blocks in a 2D cyclic fashion; namely the (i, j)-th block is assigned to $(b_r \cdot t_c + b_c)$-th thread, where the threads are organized into a $t_r \times t_c$ grid (i.e., $n_t = t_r \cdot t_c$), $b_r = \text{mod}(i, t_r)$, and $b_c = \text{mod}(j, t_c)$ (see Figure 9(b)). Since the blocks assigned to a thread are not contiguous in memory, accessing these blocks incurs some overhead. However, this offers more parallelism than the 1D layout does. We chose to use the 1D block layout if the number of columns is greater than the number of threads. Otherwise, we use the 2D cyclic layout if the number of blocks is greater than the number of threads.[2] Finally, we use a single thread to update the trailing submatrix if there are not enough blocks.

This hybrid programming paradigm obtained significant reduction in memory usage while achieving the same level of parallel efficiency as the pure MPI paradigm. As a result, in comparison to the pure MPI paradigm which failed due to the per-core memory constraint, this hybrid paradigm could

[2]In our experiments, the thread grid is as close to a square grid as possible.

Name	Application	Source	Type	Symm.	n	$\frac{nnz}{n}$	fill-ratio
tdr455k	Accelerator	Omega3P	real	Yes	$2,738,556$	41	12.3
matrix211	Fusion	M3D-C^1	real	No	$801,378$	161	9.9
cc_linear2	Fusion	NIMROD	complex	No	$259,203$	109	7.1
ibm_matick	Circuit simulation	IBM	complex	No	$16,019$	$4,005$	1.0
cage13	DNA electrophoresis	UF collection	real	No	$445,315$	17	608.5

Table I
TEST MATRIX PROPERTIES.

use more cores on each node and reduce the factorization time on the same number of nodes.

VI. PERFORMANCE RESULTS

In this section, we study the performance of the techniques proposed in this paper. We first describe our testbeds, test matrices, and experimental setups in Sections VI-A, VI-B, VI-C, respectively. Then, we present the effects of static scheduling and hybrid programming on the performance of `SuperLU_DIST` in Sections VI-D and VI-E, respectively.

A. Experimental testbeds

We conducted our experiments to examine the performance of the proposed techniques on two leading HPC systems at the National Energy Research Scientific Computing Center (NERSC). In this section, we briefly describe our experimental testbeds.

Cray-XE6 (Hopper): Hopper is a Cray-XE6 system and placed number eight on the latest Top500 Supercomputer list (June 2011). It consists of $153,216$ compute cores and 217TB of total memory, and has a peak performance of 1.28 petaflops/sec. Each compute node consists of two twelve-core AMD Magny-Cours 2.1GHz processors, giving each node 24 cores. Each Magny-Cours has two six-core Bulldozer CPUs connected by interconnect in one package, where each CPU has its own local memory controllers. Hence, it provides a NUMA architecture within each package. Each compute node has 32GB of memory with about 1.3GB of memory per core when all the cores are used on the node. These compute nodes are connected by the Cray Gemini interconnect that forms 3D torus.

IBM iDataPlex (Carver): Carver is an IBM iDataPlex system with $3,520$ processor cores. The compute node used for our experiments has two quad-core intel Xeon X5550 Nehalem 2.7GHz processors, and 24GB of memory. These nodes on Carver do not have disk, and about 4GB of the memory is used to store the system files. Hence, each core has about 2.5GB of memory when the node is fully packed. For high-performance message passing on the interconnect between the nodes, 4X QDR InfiniBand technology, with 32Gb/sec of point-to-point bandwidth, is used.

More information about our testbeds can be found at https://www.nersc.gov/systems.

B. Test matrices

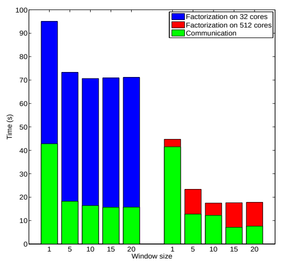
The applications of our main interests are the numerical simulations (Omega3P) to model particle accelerator cavities [2] and those (M3D-C^1 and NIMROD) to model fusion energy devices [1]. The accelerator simulation involves nonlinear eigenvalue problems for solving discretized Maxwell equations, where the solutions of the highly-indefinite linear systems are needed for the shift-invert operations. When the shift is close to an actual eigenvalue, these linear systems are close to singular and extremely difficult to solve using a preconditioned iterative method. The numerical simulation of the fusion energy devices requires the solution of linear systems of the discretized extended MHD equations, which are unsymmetric and indefinite. Besides these, we have selected two matrices from the other disciplines; one from a circuit simulation at IBM, and the other for DNA electrophoresis from the University of Florida sparse matrix collection. Table I shows the properties of our test matrices.

C. Experimental setup

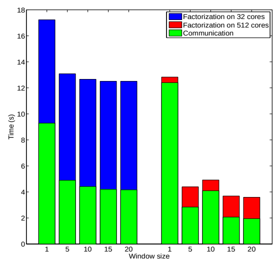
For all of our experiments in this paper, we used the default setups of `SuperLU_DIST`; i.e., we used `MC64` for static pivoting and equilibration to enhance numerical stability, a serial nested dissection algorithm of METIS to preserve the sparsity of the LU factors, and serial symbolic factorization to setup the data structures required for the numerical factorization. These serial matrix pre-processing and symbolic factorization algorithms require each MPI process to store the global coefficient matrix. It is possible to use parallel pre-processing algorithms and the parallel symbolic factorization [11] by replacing `MC64` and METIS with a simple parallel matrix equilibration [22] and a parallel nested dissection of ParMETIS [18] or PT-SCOTCH [20], respectively. However, since `SuperLU_DIST` does not perform any dynamic pivoting, `MC64` may be necessary for ill-conditioned problems. Furthermore, the matrix orderings returned by ParMETIS or PT-SCOTCH would be different using different numbers of processes, and this would make it difficult to compare the parallel performance of the proposed techniques on different numbers of processes.

D. Performance results of static scheduling

Figure 10 shows the effects of the window size n_w on the performance of static scheduling on the Cray-XE6. In the figure, the bars with the window size of one show

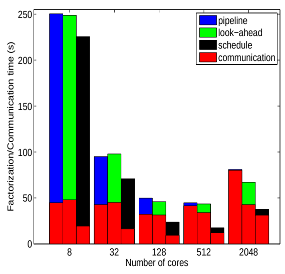


(a) tdr451k

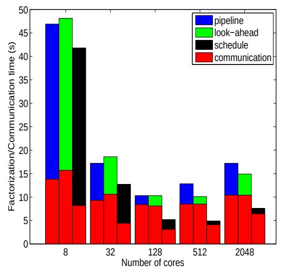


(b) matrix211

Figure 10. Effects of window size on performance of static scheduling.



(a) tdr455k



(b) matrix211

Figure 11. Factorization (MPI communication) time in seconds, with v2.5 and v3.0 on Hopper.

the numerical factorization times of the latest version 2.5 of `SuperLU_DIST`, while those with the window size of greater than one deploy the new look-ahead and static scheduling strategies in the upcoming release version 3.0. We see that the integration of look-ahead and static scheduling significantly reduced the factorization time by reducing the synchronization time and overlapping communication with computation. The improvement stagnated with the window size greater than 10.

Table II shows the performance of the static scheduling for all the test matrices with the fixed window size of $n_w = 10$. Specifically, we show the performance of look-ahead alone ("look-ahead" in the table) and the combination of look-ahead with static scheduling ("schedule"), and compare it with that of version 2.5 ("pipeline"). In the table, we clearly see that with a large number of processes, the factorization time was dominated by the communication time, which is shown in parentheses.[3] Since the communication time increased with the number of MPI processes, the pipelined factorization did not scale beyond hundreds of processes. Even though the look-ahead alone was not effective, when it was combined with the static scheduling, the communication time was significantly reduced, obtaining the speedups of up to 2.9 over the pipelined factorization time. Figure 11 shows these results visually for **tdr455k** and **matrix211**.

For **cage13**, the factorization was slower using the static scheduling on a small number of cores (e.g., 8 or 32 cores). This is mainly due to the overhead associated with the static scheduling such as irregular access to the panels and poor data locality. However, as the number of cores increases, the communication started to dominate the factorization time, and the static scheduling was able to obtain significant speedups of up to about 2.6.

We also see that our scheduling strategy could not obtain significant speedups for **ibm_matick**. This is because **ibm_matick** and its LU factors are much denser than the other test matrices. Hence, its task dependency graph is closer to a complete graph. This provides only a small potential of reducing the idle time by reordering the matrix.

[3]Integrated Performance Monitoring (IPM) was used to measure the times spent on MPI communication.

In Table III, we show the experimental results on Carver. The maximum number of nodes that a user can obtain on Carver is 64, where each node has eight cores. Hence, in order to use 512 cores, we must use all of the eight cores on each of the 64 nodes. Unfortunately, this did not provide enough memory to solve some of the linear systems on 512 cores. However, similar to the results on Hopper, even on hundreds of cores, significant speedups was obtained using the static scheduling. In Section VI-E, we study the memory usage in more details.

	Number of cores			
version	8	32	128	512
results for tdr455k				
cores/node	2	4	4	8
pipeline	195.9	65.7	39.4	OOM
schdule	172.3	47.0	17.8	OOM
results for matrix211				
cores/node	8	8	8	8
pipeline	46.9	14.7	10.1	14.4
schedule	31.8	7.8	5.7	13.4
results for cc_linear2				
cores/node	8	8	8	8
pipeline	27.4	8.5	5.7	6.3
schdule	18.4	4.9	2.5	2.7
results for ibm_matick				
cores/node	4	4	4	8
pipeline	27.0	7.12	2.1	OOM
schdule	25.1	6.77	2.1	OOM
results for cage13				
cores/node	1	2	2	8
pipeline	5104.6	1322.3	335.4	OOM
schdule	7041.2	1316.8	320.2	OOM

Table III
FACTORIZATION TIME IN SECONDS WITH V2.5 AND V3.0 ON CARVER.

E. Performance results of hybrid programming

Table IV shows the results of the hybrid programming paradigm, where we used different numbers of MPI processes and OpenMP threads on 16 compute nodes of Hopper.

	Number of cores				
version	8	32	128	512	2048
results for tdr455k					
cores/node	1	8	8	8	4
pipeline	250.3 (44.8)	95.0 (42.8)	49.8 (32.2)	44.7 (41.5)	81.0 (80.1)
look-ahead(10)	248.7 (48.0)	97.9 (45.1)	45.9 (31.5)	43.5 (34.1)	67.1 (42.8)
schdule	225.5 (19.4)	70.7 (16.4)	23.7 (9.3)	17.5 (12.2)	37.7 (31.3)
results for matrix211					
cores/node	8	24	24	24	8
pepeline	46.9 (13.8)	17.2 (9.3)	10.3 (8.4)	12.8 (12.4)	17.2 (17.1)
look-ahead(10)	48.1 (15.7)	18.6 (10.6)	10.3 (8.1)	10.1 (8.5)	14.9 (10.4)
schedule	41.8 (8.2)	12.7 (4.4)	5.2 (3.1)	4.9 (4.1)	7.6 (6.4)
results for cc_linear2					
cores/node	8	24	24	24	8
pipeline	30.9 (29.6)	12.3 (7.2)	7.6 (6.5)	6.8 (6.6)	7.9 (7.9)
schedule	24.3 (17.8)	7.5 (2.3)	3.6 (2.4)	2.3 (2.0)	2.7 (2.5)
results for ibm_matick					
cores/node	8	8	8	8	4
pipeline	46.9 (12.6)	14.9 (4.8)	7.2 (5.5)	5.4 (5.0)	5.2 (5.1)
schedule	46.4 (13.3)	12.5 (3.1)	7.0 (5.0)	5.0 (4.6)	4.8 (4.6)
results for cage13					
cores/node	1	4	4	4	4
pipeline	6798.9 (425.8)	1986.4 (287.5)	481.4 (134.9)	139.5 (61.6)	124.5 (107.7)
schedule	8412.5 (600.5)	2085.6 (241.7)	438.6 (86.8)	116.0 (34.6)	47.5 (21.5)

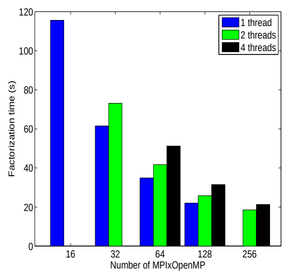
Table II
FACTORIZATION (MPI COMMUNICATION) IN SECONDS, WITH V2.5 AND V3.0 ON HOPPER.

The look-ahead window size is fixed at $n_w = 10$. In the table, "time (s)" is the numerical factorization time in seconds. In addition, next to "mem (GB)," we show the total memory allocated by `SuperLU_DIST` for the data structures storing the distributed LU factors and for the communication buffers used during the numerical factorization in Gigabytes. This value does not change with the increase in the number of MPI processes or OpenMP threads since the serial pre-processing is used. Finally, under "mem (GB)," we show the three memory statistics mem, $mem_1 + mem_2$, where mem is the total high watermark of the memory allocated by `SuperLU_DIST`, mem_1 is the total memory usage including the system memory before the factorization, and $mem_1 + mem_2$ is the usage after the factorization.[4]

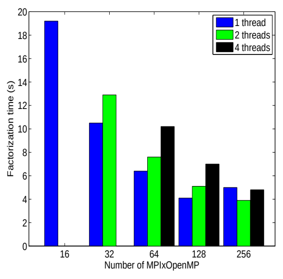
First, we clearly see that due to the serial algorithms used by the default setups, mem increased almost proportionally to the number of MPI processes. We also see that $mem_1 + mem_2$ was significantly greater than mem. This is mainly because on Hopper, all the libraries are statically linked by default and this leads to a large executable file. The hybrid programming paradigm reduces these memory bottlenecks using OpenMP threads in place of MPI processes. As a result, the hybrid program could effectively use more cores on each node, whereas the pure MPI program failed due to the per-core memory constraint.[5]

Furthermore, we see that the best time for each matrix with the fixed node count of 16 was always obtained by the hybrid paradigm. For example, the best time of the hybrid paradigm on **cage13** was about 2.2 times faster than that of the pure MPI paradigm (cf., 845.3 seconds with 64×1 while 377.2 seconds with 64×4). This clearly shows that the hybrid paradigm was able to better utilize the resources available on the compute nodes.

Finally, when the same number of cores is used, the factorization was faster using the pure MPI paradigm on a small number of cores. However, on a large number of cores, the hybrid paradigm could avoid the expensive message passing among the cores on the same NUMA node and obtained a small speedup over the pure MPI paradigm (e.g., 3.9 seconds with 128×2 while 5.0 seconds with 256×1 for **matrix211**). Figure 12 shows these timing results visually for **tdr455k** and **matrix211**.



(a) tdr455k



(b) matrix211

Figure 12. Results of hybrid programming using 16 nodes of Hopper.

Table V shows the results of the hybrid programming on Carver. The behavior of the codes was similar to those

[4] The memory usage was obtained by reading the system file /proc/(pid)/status.

[5] The program may fail at a serial bottleneck before the factorization (e.g., serial equilibration or symbolic factorization).

	tdr455k			matrix211			cage13		
MPI×Thread	time (s)	mem (GB); 23.3		time (s)	mem (GB); 5.4		time (s)	mem (GB); 43.3	
16×1	115.6	45.4,	$75.2 + 0.8$	19.2	15.4,	$56.3 + 0.5$	3943.7	106.3,	$91.5 + 1.1$
32×1	61.5	81.9,	$128.0 + 1.1$	10.5	30.7,	$106.6 + 1.1$	1743.3	169.5,	$141.7 + 1.8$
16×2	73.1	45.4,	$75.2 + 1.5$	12.9	15.4,	$56.3 + 1.1$	2221.0	106.3,	$91.5 + 1.8$
64×1	34.9	163.8,	$232.0 + 2.0$	6.4	61.5,	$207.3 + 2.9$	845.3	294.3,	$243.7 + 3.3$
32×2	41.7	81.9,	$128.1 + 2.3$	7.6	30.7,	$106.6 + 2.4$	1014.1	169.5,	$141.7 + 3.1$
16×4	51.2	45.4,	$75.2 + 2.8$	10.2	15.4,	$56.3 + 2.4$	1297.1	106.3,	$91.5 + 3.1$
128×1	22.0	327.6,	$441.1 + 6.9$	4.1	122.9,	$410.5 + 5.0$	$--$	OOM,	OOM
64×2	25.8	163.8,	$232.0 + 4.6$	5.1	61.5,	$207.3 + 5.5$	567.2	294.3,	$243.7 + 5.8$
32×4	31.5	81.9,	$128.1 + 4.8$	7.0	30.7,	$106.6 + 4.9$	750.0	169.5,	$141.7 + 5.7$
16×8	41.7	45.4,	$75.2 + 5.3$	9.1	15.4,	$56.3 + 5.0$	905.9	106.3,	$91.5 + 5.6$
256×1	$--$	OOM,	OOM	5.0	245.9,	$830.9 + 8.2$	$--$	OOM,	OOM
128×2	18.5	327.6,	$441.1 + 12.0$	3.9	122.9,	$410.5 + 10.1$	$--$	OOM,	OOM
64×4	21.3	163.8,	$232.0 + 9.7$	4.8	61.5,	$207.3 + 4.6$	377.2	169.5,	$243.7 + 10.9$

Table IV
RESULTS OF HYBRID PROGRAMMING USING 16 NODES OF HOPPER.

on Hopper. The only significant difference was that much less system memory was required on Carver. This is mainly because on Carver, some of the libraries are dynamically linked, and the executable files is usually much smaller than that on Hopper.

VII. CONCLUSION

We studied two strategies to enhance the parallel performance of `SuperLU_DIST` on modern multicore architectures. The first strategy schedules independent tasks as soon as possible to shorten the critical path. The experimental results demonstrated that the parallel factorization with this new scheduling strategy is nearly three times faster than the previous pipelined factorization. The second strategy uses the hybrid programming to overcome per-core memory constraint and fully utilize the node-level parallelism on a NUMA manycore architecture. We incorporated light-weight OpenMP threads in each MPI process to update independent blocks of the trailing submatrix. This hybrid programming could reduce the memory usage significantly, while achieving the same level of parallel efficiency as a pure MPI code. As a result, in comparison to the pure MPI paradigm, the hybrid paradigm utilized more cores on each node and reduced the factorization time on the same number of nodes.

In order for our static scheduling scheme to capture the different computational costs of the panel factorization tasks, we have assigned weights on the edges in our task dependency graphs (e.g., based on the size of the diagonal block). Furthermore, the MPI processes are currently assigned to supernodal blocks before the static scheduling. It might be beneficial to consider the process-assignment during the static scheduling such that the leaf-nodes are scheduled in a round-robin fashion according to the processes assigned to them. The motivation was to allow multiple processes to factorize different leaf-nodes in parallel. We have investigated these approaches, but currently, we have not observed significant improvements over the strategies described in Section IV.

We currently use the hybrid programming paradigm only for the trailing submatrix update. We are considering how we can apply the hybrid paradigm for the panel factorization and reduce the message-passing overhead.

Finally, we plan to extend our memory profiling studies of the hybrid programming paradigm. For instance, we have not measured the amount of memory internally allocated by MPI during the numerical factorization. These measurements may show more significant advantages of the hybrid programming paradigm over the pure MPI paradigm.

ACKNOWLEDGEMENT

This research was supported in part by the Director, Office of Science, Office of Advanced Scientific Computing Research of the U.S. Department of Energy under Contract No. D-AC02-05CH11231. We gratefully thank Esmond Ng and David Skinner for helpful discussions.

REFERENCES

[1] Center for Extended MHD Modeling (CEMM). http://w3.pppl.gov/cemm/.

[2] Community Petascale Project for Accelerator Science and Simulation (ComPASS). https://compass.fnal.gov.

[3] A. Aho, M. Garey, and J. Ullman. The transitive reduction of a directed graph. *SIAM J. Comput.*, 1:131–137, 1972.

[4] P. Amestoy, I. Duff, J.-Y. L'excellent, and X. Li. Analysis and comparison of two general sparse solvers for distributed memory comoputers. *ACM Transactions on Mathematical Software*, pages 388–421, 2001.

[5] M. Baertschy and X. S. Li. Solution of a three-body poblem in quantum mechanics. In *Proceedings of SC2001: High Performance Networking and Computing Conference*, Denver, Colorado, November 10–16 2001.

	tdr455k			matrix211		
MPI×Thread	time (s)	mem (GB); 23.2		time (s)	mem (GB); 5.4	
16×1	92.4	41.0	$26.8 + 0.7$	15.1	15.4	$7.8 + 0.6$
32×1	51.1	81.9	$32.2 + 1.3$	9.0	30.7	$12.0 + 1.0$
16×2	53.1	41.0	$27.8 + 1.1$	9.9	15.4	$8.8 + 0.9$
64×1	28.8	163.8	$41.3 + 2.6$	5.5	61.5	$18.7 + 2.3$
32×2	28.5	81.9	$32.1 + 2.0$	5.8	30.7	$12.0 + 1.7$
16×4	32.2	41.0	$26.8 + 1.7$	6.9	15.4	$8.8 + 1.5$
128×1	$--$	OOM	OOM	6.0	123.0	$34.3 + 4.5$
64×2	19.3	163.8	$41.3 + 3.8$	3.7	61.5	$18.7 + 3.5$
32×4	20.7	81.9	$32.2 + 3.2$	4.3	30.7	$12.0 + 3.0$

Table V
RESULTS OF HYBRID PROGRAMMING USING 16 NODES OF CARVER.

[6] J. Dongara, P. Beckman, and et.al. The international exascale software roadmap. *International Journal of High Performance Computer Applications*, 25, 2011.

[7] I. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Anal. Appl.*, 20(4):889–901, 1999.

[8] S. C. Eisenstat and J. W. H. Liu. Exploiting structural symmetry in sparse unsymmetric symbolic factorization. *SIAM J. Matrix Anal. Appl.*, pages 13:202–211, 1992.

[9] M. Faverge, X. Lacoste, and P. Ramet. A NUMA aware scheduler for a parallel sparse direct solver. In *Proceedings of the international workshop on parallel matrix algorithms and applications (PMAA)*, 2008.

[10] J. R. Gilbert. Predicting structure in sparse matrix computations. *SIAM J. Matrix Analysis and Applications*, 15:62–79, 1994.

[11] L. Grigori, J. W. Demmel, and X. S. Li. Parallel symbolic factorization for sparse lu with static pivoting. *SIAM J. Sci. Comput.*, 29:1289–1314, 2007.

[12] L. Grigori and X. S. Li. A new scheduling algorithm for parallel sparse LU factorization with static pivoting. In *Proceedings of ACM/IEEE conference on supercomputing*, pages 1–18, 2002.

[13] A. Gupta. WSMP: Watson Sparse Matrix Package, Part I - direct solution of symmetric sparse systems. Technical Report RC 21886 (98462), IBM Research, 2000.

[14] A. Gupta. WSMP: Watson Sparse Matrix Package, Part II - direct solution of general sparse systems. Technical Report RC 21888 (98472), IBM Research, 2000.

[15] P. Hénon, P. Ramet, and J. Roman. PaStiX: A high-performance parallel direct solver for sparse symmetric definite systems. *Parallel Computing*, 28:301–321, 2002.

[16] P. Hénon, P. Ramet, and J. Roman. On using an hybrid MPI-Thread programming for the implementation of a parallel direct solver on a network on SMP nodes. In *Proceedings of the international conference on parallel processing and applied mathematics*, 2005.

[17] J. D. Hogg, J. K. Reid, and J. A. Scott. Design of a multicore sparse cholesky factorization using dags. *SIAM J. Sci. Comput.*, 32:3627–3649, 2010.

[18] Karypis Lab, Digital Technology Center, Department of Computer Science and Engineering, University of Minesota. Family of graph and hypergraph partitioning software. http://glaros.dtc.umn.edu/gkhome/views/metis.

[19] J. Kurzak and J. Dongarra. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. lapack working note 178. Technical report, University of Tennessee, 2006.

[20] Laboratoire Bordelais de Recherche en Informatique (LaBRI). SCOTCH - software package and libraries for graph, mesh and hypergraph partitioning, static mapping, and parallel and sequential sparse matrix block ordering. http://www.labri.fr/perso/pelegrin/scotch/.

[21] X. S. Li and J. W. Demmel. Making sparse Gaussian elimination scalable by static pivoting. In *Proceedings of SC98: High Performance Networking and Computing Conference*, 1998.

[22] X. S. Li and J. W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29(2):110–140, June 2003.

[23] X. S. Li, J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Anal. Appl.*, 20:720:755, 1999.

[24] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Appl.*, 1:134–172, 1990.